Samuli Määttä

MOBIILISOVELLUKSEN TIETOKANTAKIRJASTON UUDISTAMINEN

MOBIILISOVELLUKSEN TIETOKANTAKIRJASTON UUDISTAMINEN

Samuli Määttä
Opinnäytetyö
Kevät 2020
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehitys

Tekijä: Samuli Määttä
Opinnäytetyön nimi suomeksi: Mobiilisovelluksen tietokantakirjaston uudistaminen
Opinnäytetyön nimi englanniksi: Renewal of mobile application’s database library
Työn ohjaajat: Mika Mustonen, Veikko Tapaninen
Työn valmistumislukukausi ja -vuosi: Kevät 2020
Sivumäärä: 49

Tämän opinnäytetyön tavoitteena oli tutkia ja tehdä esimerkkitoteutus, kuinka Room-tietokantakirjasto voisi korvata vanhan SugarORM-tietokantakirjaston Polarin Flow App Android -sovelluksessa. Työ oli tarpeellinen Polarin mobiilikehittäjille, sillä vanhentunut tietokantakirjasto ja sen arkkitehtuuri sovelluksessa vaikeuttavat kehitystä.

Aiemmin Polarille tehdyssä yrityslähtöisen tuotekehitysprojektin vertailussa, Room-tietokantakirjasto osoittautui parhaaksi valinnaksi sovelluksen uudeksi tietokantakirjastoksi. Sovellus oli jo valmiiksi tuttu, joten siihen tutustuminen ei vienyt aikaa.

Työ aloitettiin miettimällä Polarin asiantuntijoiden kanssa sovelluksen uutta arkkitehtuuria tietokantaan liittyen, sekä mitä turhaa nykyisessä tietokannassa on. Työn aikana pidettiin arkkitehtuuripalavereita ja koodikatselmointeja.

Tuloksena saatiin aikaan toimiva esimerkkitoteutus Room-tietokantakirjaston käyttöönotosta, jossa on käytössä uusi palvelupohjainen arkkitehtuuri, missä tietokantaoperaatiot on abstraktoitu palveluiden ja erillisten DAO-luokkien sisään. Sovellukseen tehtiin myös käyttäjälogiikan refaktorointia paremmin sopivaksi uuteen arkkitehtuuriin. Tietokannan viite-eheys varmistettiin relaatioilla. Esimerkkitoteutukseen tehtiin myös tietokannan salaus sekä yksikkötestejä tietokannan testaamiseen. Tietokantamigraatioiden tekemisestä dokumentoitiin Polarin sisäiseen Confluence-wikiin. Työ tehtiin Kotlin-ohjelmointikielellä sekä vanhaa koodia refaktoroitiin käyttämään Kotlinia, sallien Kotlinin vuorottaisrutiinien käytön. Työ tukee myös tulevaisuudessa käyttöönotettavaa MVVM-suunnittelumallia.

Esimerkkitoteutus saatiin koodikatselmointiin, josta se voi hyväksymisen jälkeen edetä testaukseen ja tuotantoon.

Asiasanat: tietokanta, mobiilisovellus, Android, SQL, Room, Java, Kotlin

ABSTRACT

Oulu University of Applied Sciences
Degree programme in Information Technology

Author: Samuli Määttä
Title of thesis: Renewal of mobile application's database library
Supervisors: Mika Mustonen, Veikko Tapaninen
Term and year when the thesis was submitted: Spring 2020
Pages: 49

Goal of this thesis work was to research and do a proof of concept implementation of how Room database library would replace old SugarORM database library in Polar's Flow App Android mobile application. Work was requested by Polar's mobile developers, as outdated database library and its implementation in application were slowing down development process.

I had previously done comparison of database libraries as company-oriented product development course. I found out that Room database library is the best choice as new library. Application was also familiar to me so getting into it didn't take time.

Work began with figuring new architecture related to database with mobile developers of Polar. It was also discussed what unneeded data were in old database. While work was in progress, architectural meetings and code reviews were held.

There were no major difficulties during work. As a result, working example implementation of migrating to Room was achieved. Implementation consisted new architectural improvements, where database is abstracted inside services and database operations are managed by Data Access Objects. Some user logic was also refactored to meet new architecture. Referential integrity of database was accomplished with usage of relationships. Example implementation also consisted of encrypting database with Android Keystore and implementing unit tests to test database. Documentation of Room migrations was also made to Polar's internal Confluence wiki. New code was written in Kotlin and some old code was refactored to Kotlin to allow use of coroutines. This work also supports new MVVM-pattern which can be taken use in future.

Pull request of example implementation was opened and code review is currently in progress. After changes are approved, implementation can proceed to testing and be merged to final product.

Keywords: database, mobile application, Android, SQL, Room, Java, Kotlin

ALKULAUSE

Aluksi tahdon kiittää Polarin linjaesimiestä Mika Mustosta ja OAMK:n lehtoria Veikko Tapanista tämän työn ohjaamisesta sekä Polarin mobiilikehittäjiä teknisestä avusta ja näkökulmista. Tämä työ syvensi hyvin osaamista Android-mobiilisovelluskehityksestä sekä tietokannoista.

Oulussa 10.1.2020

Samuli Määttä

SISÄLLYS

SANASTO

AES	Advanced Encryption Standard, vahva lohkosalausmenetelmä.
APK	Android Application Package, Android-sovelluksen asennuspaketti, josta Android-sovellus voidaan asentaa Android-laitteeseen.
BLE	Bluetooth Low Energy, langaton tiedonsiirtotekniikka.
BLOB	Binary Large OBject, SQLitessä käytössä oleva tietotyyppi, johon voidaan tallentaa binääridataa.
Builder	Olio-ohjelmoinnin suunnittelumalli, jolla pyritään helpottamaan olion luomisprosessia siinä tapauksessa, jos olio voi ottaa vastaan paljon erilaisia parametreja tai niiden yhdistelmiä.
Coroutine	Vuorottaisrutiini, Kotlinin ominaisuus asynkronisen ohjelmakoodin kirjoittamiseen.
CWAC-SafeRoom	Avoimen lähdekoodin kirjasto, joka toimii siltana Roomin ja SQLCipher-laajennuksen välillä.
DAO	Data Access Object, DAO-luokista luotava olio, jolla tietokantaa käsitellään.
Foreign key	Viiteavain, käytetään relaatiotietokannoissa viittaamaan toisissa tietokantatauluissa oleviin pääavaimiin, varmistaen viite-eheyden.
Gradle	Koontityökalu, jota käytetään liitännäisenä Android Studiossa.
MVVM	Model–View–ViewModel, suunnittelumalli, jolla erotetaan käyttöliittymä muusta sovelluslogiikasta.

Null	Tyhjäarvo, arvo, jota ei ole määritelty ja joka voi olla mitä tahansa tai ei mitään.
ORM	Object-Relational Mapping, tekniikka olio-ohjelmoinnin olioiden muuttamiseen tietokannan ymmärtämään muotoon.
Polar Flow	Polar Electron ylläpitämä ekosysteemi, jossa käyttäjän data tallennetaan ja synkronoidaan rannelaitteen, mobiilisovellusten ja verkkopalvelun välillä.
Primary key	Pääavain, käytetään relaatiotietokannoissa yksilöimään tietokantatalun rivejä.
Protocol buffer	Googlen kehittämä serialisointitekniikka, joka on laajalti käytössä Polar Flow -ekosysteemissä.
Pull request	Vetopyyntö, käytetään yhdistämään kaksi koodihaaraa toisiinsa. Kehittäjät näkevät muutokset, mitä halutaan yhdistää ja voivat kommentoida, hyväksyä tai merkitä työn keskeneräiseksi.
Relaatiotietokanta	Relaatiomalliin perustuva tietokanta.
Room	Room Persistence Library, ORM-tekniikkaan perustuva moderni tietokantakirjasto Androidille.
Schema	Skeema, relaatiotietokannan rakenne.
Singleton	Olio-ohjelmoinnin suunnittelumalli, jossa luokasta voidaan tehdä vain yksi instanssi.
SQL	Structured Query Language, kyselykieli, jolla voidaan lisätä, hakea, muuttaa tai poistaa tietoa relaatiotietokannasta.
SQLite	Relaatiotietokannan hallintajärjestelmä.

SugarORM	ORM-tekniikkaan perustuva vanhentunut tietokantakirjasto Androidille.

1 JOHDANTO

Tämän opinnäytetyön tilaajana toimii Polar Electro Oy. Polar Electro Oy on Kempeleessä vuonna 1977 perustettu urheilukellojen, aktiivisuusrannekkeiden, sykemittareiden ja pyöräilytietokoneiden valmistaja. Opinnäytetyön kohde on Polar Flow Android -mobiilisovelluksen tietokantakirjaston uudistaminen. Työlle oli kehittäjien puolelta tarvetta, sillä vanhentunut tietokantakirjasto ja sen arkkitehtuuriset ratkaisut sovelluksessa vaikeuttivat uusien ominaisuuksien kehitystä ja hankaloittivat sovelluksen ylläpitoa. Sovelluksessa tietokantakirjastoa käytetään helpottamaan tietokannan hallintaa. Tietokantaa käytetään tiedon tallentamiseen paikallisesti. Tietokanta sisältää esimerkiksi harjoitusdataa, aktiivisuusdataa, unidataa ja käyttäjän asetuksia.

Olin tehnyt ennen tätä opinnäytetyötä yrityslähtöisenä tuotekehitysprojektina tietokantakirjastojen vertailun, jossa Room-tietokantakirjasto osoittautui parhaaksi valinnaksi korvaamaan vanha SugarORM-tietokantakirjasto. Olin myös työskennellyt ennen tämän sovelluksen parissa, joten sovellus oli entuudestaan tuttu.

Työn tavoite oli tehdä soveltuvuusselvitys eli osoittaa toteutus Room-tietokantakirjaston käyttöönotosta toteuttamiskelpoiseksi. Valitsin refaktoroitavaksi osaksi sovelluksen user-preferences-osuuden. Sovellukseen oli jo hieman koeponnistettu service-pohjaista arkkitehtuuria, jossa verkko-operaatioita varastoidaan palveluiden sisään. Tässä työssä tietokantaoperaatiot varastoidaan myös palveluihin ja erillisiin DAO-luokkiin. Palvelut tehdään myös tietokanta-agnostisiksi, joten tulevaisuudessa tietokantatoteutuksen muuttaminen on huomattavasti helpompaa. Työssä toteutettiin myös tietokannan salaaminen, dokumentoitiin tietokantamigraatioiden tekemisestä ja tehtiin yksikkötestejä tietokannan testaamiseen.

2 ANDROID

2.1 Android-järjestelmä

Android-järjestelmällä tarkoitetaan Android-ohjelmistopinoa, joka koostuu käyttöjärjestelmästä, väliohjelmistoista ja käyttäjän perusohjelmista. Ohjelmistopinon alin kerros koostuu Linux-käyttöjärjestelmäytimestä, jota on muokattu esimerkiksi muistin- ja virranhallinnan osalta. (1.)

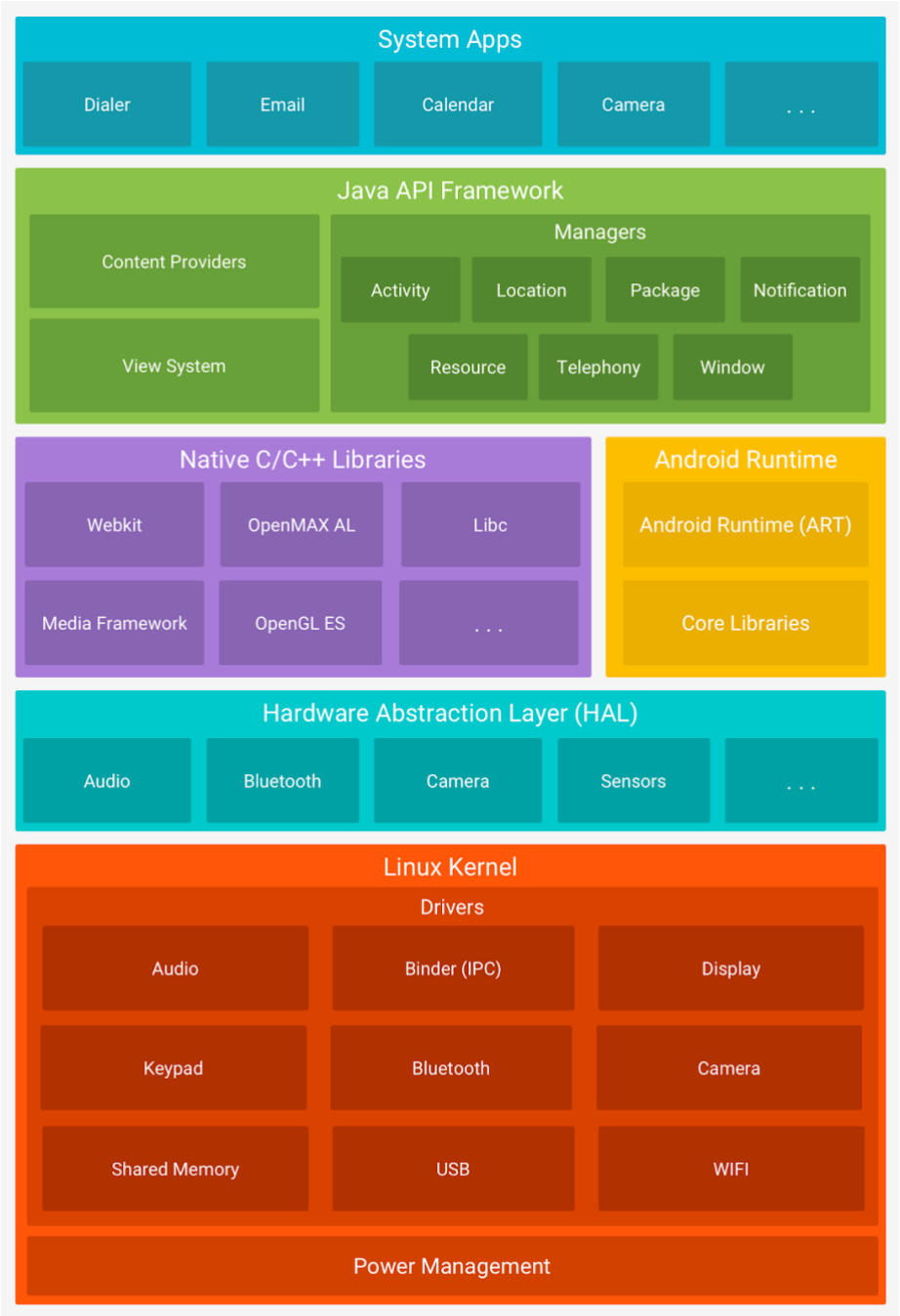
Seuraavana kerroksena on Hardware Abstraction Layer (HAL), joka toimii rajapintana laitekomponenttien ja ohjelmistokirjastojen välillä. HAL:n avulla laitevalmistajat voivat toteuttaa toiminnallisuudet esimerkiksi kamera- ja Bluetooth-komponenteille niin, että ylemmän tason Java API Framework pysyy ennallaan. (1.)

Android-sovellukset suoritetaan käyttäen Android Runtime (ART) -ajoympäristöä, jonka tehtävä on kääntää DEX-tavukoodia konekielisiksi käskyiksi, ART korvasi edellisen Dalvik-ajoympäristön Android-versiossa 5.0. (2.)

Android-järjestelmässä on myös kokoelma natiiveja C/C++-kirjastoja, kuten OpenGL ES -grafiikkarajapinta, WebKit-selainmoottori ja SQLite-relaatiotietokanta. Niitä voidaan käyttää suoraan ylemmän tason Java API Frameworkin kautta. (1).

Android-järjestelmää käyttäviä laitteita ovat esimerkiksi puhelimet, tabletit ja televisiot. Ensimmäinen kaupallinen Android-laite oli syyskuussa 2008 julkaistu HTC Dream -älypuhelin. (3.)

Kuva 1 esittää Android-ohjelmistopinoa.



KUVA 1. Android-ohjelmistopino (1).

2.2 Android-sovelluskehitys

Android-sovelluskehityksellä tarkoitetaan sovellusten kehittämistä Android-alustalle. Android-sovelluksia kehitetään yleensä Kotlin- ja Java-ohjelmointikielillä käyttäen JDK:ta (Java Development Kit) ja Android SDK:ta (Software Development Kit). Sovelluksia voidaan myös kehittää C++-kielellä käyttäen Android NDK:ta (Native Development Kit), sekä erilaisilla cross-platform työkaluilla, joilla voidaan tehdä yksi sovellus, joka toimii Androidin lisäksi muilla

alustoilla. Tosin näillä cross-platform työkaluilla tehdyt sovellukset yleensä häviävät suoritusnopeudessa ja käyttökokemuksessa natiiveille sovelluksille. Tunnettuja Android-kehitysympäristöjä ovat esimerkiksi Android Studio ja Eclipse. (4.)

Valmis Android-sovellus voidaan jakaa APK-tiedostona, josta sovelluksen saa asennettua Android-laitteelle. APK-tiedoston luominen onnistuu Android-kehitysympäristöillä, kuten Android Studiolla. APK-tiedosto on pakattu tiedosto, joka sisältää ohjelmakoodista käännetyt DEX-tiedostot, alustakohtaista koodia eri prosessoriarkkitehtuureille, sovelluksen resurssitiedostot ja muuta metadataa. (5.) Sovelluksia saa ladattua internetistä. Virallinen paikka ladata sovelluksia on Googlen Google Play -sovelluskauppa. Sovelluksia saa ladattua myös kolmansien osapuolien sovelluskaupoista ja verkkosivuilta. (6.) Myös muissa järjestelmissä oleva APK-paketti saadaan asennettua Android-laitteelle USB-yhteyden kautta käyttäen Android Debug Bridgeä (ADB) (5).

Tässä työssä käytetään Kotlin- ja Java-ohjelmointikieliä, JDK:ta, Android SDK:ta ja Android Studiota.

2.2.1 Java

Java on Sun Microsystemsin kehittämä ohjelmistoalusta ja ohjelmointikieli. Se julkaistiin vuonna 1995. Java on oliopohjainen, staattisesti tyypitetty, imperatiivinen, käännetty kieli. Java käännetään tavukoodiksi, jota voidaan suorittaa millä tahansa alustalla, jolla on Java-virtuaalikone (JVM). (7.)

2.2.2 Kotlin

Kotlin on JetBrainsin kehittämä ohjelmointikieli. Kotlin on suunniteltu täysin yhteensopivaksi Javan kanssa. Sillä pyritään ratkaisemaan Javan heikkouksia. Kotlinilla ohjelmoidessa tarvitaan vähemmän itseään toistavaa ohjelmakoodia (boilerplate code) kuin Javalla. Kotlinin ominaisuuksiin kuuluu myös parempi null-arvojen hallinta kuin Javassa. Tämä vähentää ohjelmointivirheitä ja niistä aiheutuvia sovelluksen kaatumisia sekä muuta virheellistä toimintaa. (8.) Kotlin on myös Googlen suosittelema kieli Android-kehitykseen (9).

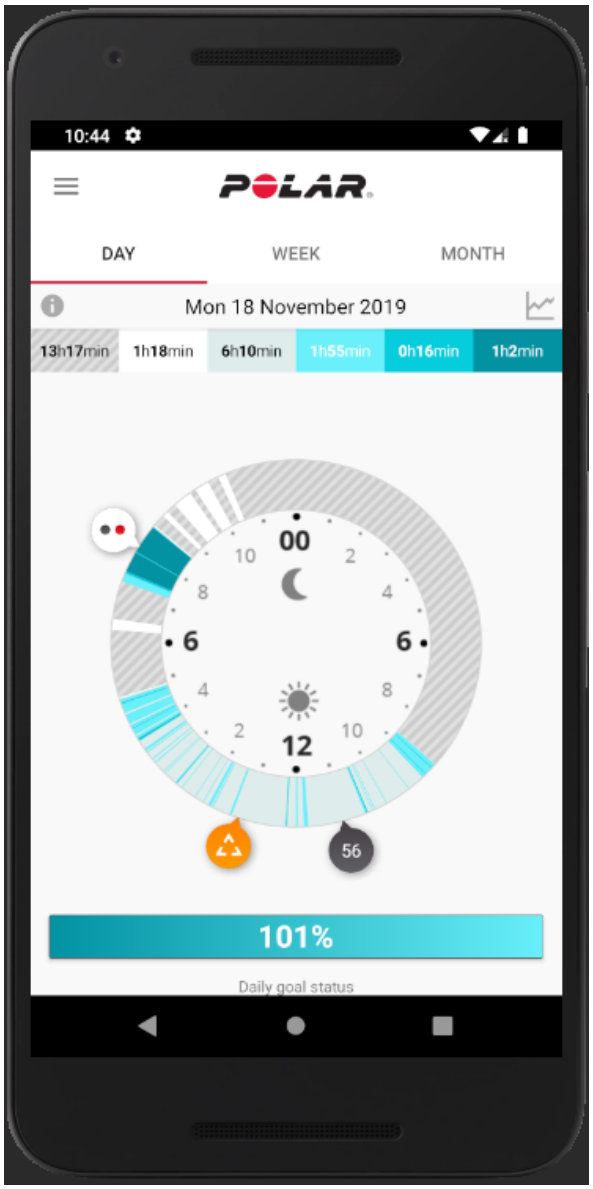
2.3 Polar Flow App -sovellus

Polar Flow App -sovellus on Polar Electro Oy:n kehittämä mobiilisovellus Android- ja iOS -alustoille. Sovellus on tarkoitettu käytettäväksi Polarin valmistamien urheilukellojen, fitnessmittareiden ja aktiivisuusmittareiden kanssa. Jokaisella käyttäjällä on yksilöllinen käyttäjätili Polar Flow -ekosysteemissä. Käyttäjät rekisteröityvät Flow-palveluun joko mobiilisovelluksella tai verkkopalvelun kautta ja parittavat Polar-laitteen sovelluksen kanssa. Polar-laitteen ja mobiilisovelluksen välillä tiedot synkronoidaan käyttäen BLE-tekniikkaa. (10.)

Sovelluksessa voi esimerkiksi analysoida tehtyjä harjoituksia, luoda harjoitusohjelmia ja asettaa harjoitustavoitteita. Harjoituksille on olemassa eri harjoitusprofiileita yli 130 kappaletta, joita voi itse muokata. Sovelluksella voi seurata omaa aktiivisuutta ympäri vuorokauden. Jokaisesta päivästä näkee yhteenvedon, joka perustuu aktiivisuuden ja sykkeen seurantaan. Sovelluksesta näkee myös, kuinka paljon tiettynä päivänä on kulunut kaloreita ja kuinka monta askelta on otettu. Myös unen kestoa ja laatua on mahdollista seurata sovelluksesta. Android-sovellus on julkaistu Google Play -sovelluskaupassa. Sovellus on ilmainen ja sitä kehitetään sekä päivitetään jatkuvasti. (11.) Kuva 2 esittää päänäkymää Polar Flow App Android -sovelluksessa.

Sovellus on pääosin tehty Java-ohjelmointikielellä. Tosin kehittäjät ovat siirtymässä Kotliniin uusia ominaisuuksia kehittäessä. Sovelluksessa on myös C/C++-kirjastoja, kuten Polarin oma matematiikkakirjasto.

Tässä työssä refaktoroidaan Polar Flow App Android -sovelluksen user preferences-osuus käyttämään Room-tietokantakirjastoa tiedon tallentamiseen. Tämä user-preferences-osuus sisältää tietoa, millä kielellä rannelaite on, tiedon, käytetäänkö SI-järjestelmää vai brittiläistä yksikköjärjestelmää, tiedon, käytetäänkö 12- vai 24-tunnin aikaformaattia, sekä tiedon viikon alkamispäivästä.



KUVA 2. Polar Flow App Android 4.3.1 -sovelluksen päänäkymä emuloidussa Nexus 5X Android 9.0 -laitteessa.

3 TIETOKANTA

3.1 Tietokanta

Tietokannalla tarkoitetaan tietorakennetta, johon voidaan varastoida tietoa. Tietokannat voivat perustua eri malleihin kuten hierarkkiseen malliin, oliopohjaiseen malliin tai relaatiomalliin. (12.) Tässä työssä käytetään relaatiotietokantaa, joka pohjautuu relaatiomalliin, jossa tietoa kuvataan matemaattisina relaatioina. Relaatiotietokanta koostuu tauluista, jotka koostuvat sarakkeista (column) ja riveistä (row). Taulujen välille voidaan tehdä viittauksia viiteavaimilla (foreign key) pääavaimiin (primary key). (13.)

Relaatiotietokantaan voidaan tehdä kyselyitä SQL-kielellä (12).

3.2 SQLite

SQLite on avoimeen lähdekoodiin pohjautuva relaatiotietokannan hallintajärjestelmä. Se eroaa muista relaatiotietokannoista, kuten MySQL-tietokannasta niin, että SQLite ei tarvitse erillistä tietokantapalvelinta, vaan luku- ja kirjoitusoperaatiot suoritetaan suoraan tiedostojärjestelmällä olevaan tietokantatiedostoon. Sen toteutuksessa on pyritty pieneen kokoon: kaikkien ominaisuuksien kanssa kirjaston koko voi olla vähemmän kuin 600 kilotavua. SQLite noudattaa ACID (Atomicity, Consistency, Isolation, Durability) -periaatetta, jolla turvataan tietokannan tietojen eheys missä tahansa tilanteessa, kuten esimerkiksi järjestelmän kaatuessa tai sähkökatkon sattuessa. (14.)

SQLite on käytössä muun muassa Android-, iOS-, macOS- ja Windows 10 -käyttöjärjestelmissä, sekä Firefox-, Chrome- ja Safari-verkkoselaimissa (15). SQLite julkaistiin vuonna 2000 ja sen kehittäjät aikovat ylläpitää sitä vuoteen 2050 asti (14).

SQLite-tietokantaa käsitellään SQL-kyselyillä. Kyselyillä voidaan hakea tietoa, lisätä tietoa, muokata tietoa tai poistaa sitä sekä luoda ja muokata tietokannan tauluja ja relaatioita. SQLite ei kuitenkaan tue ihan kaikkia SQL-kielen mukaisia kyselyitä, kuten “ALTER TABLE DROP COLUMN” -kyselyä, jolla voidaan poistaa annetusta tietokannan taulusta sarake. (16.)

3.3 Android-tietokantakirjasto

Androidissa on valmiina rajapinta SQLitelle, jota voidaan käyttää ilman varsinaista tietokantakirjastoa. Kuitenkin SQLiten käyttäminen ilman tietokantakirjastoa on työläämpää ja virheherkempää. Google suosittelee käytettäväksi Room Persistence Library -tietokantakirjastoa. (17.) Roomin lisäksi muita tunnettuja tietokantakirjastoja on muun muassa: SugarORM, Realm, DBFlow, GreenDAO ja ORMLite. Näistä kaikki paitsi Realm toimivat abstraktiokerroksena Androidin SQLiten päällä, tavoitteena tehdä tietokannan hallitsemisesta helpompaa. Realm on poikkeus, sillä se ei käytä Androidin SQLite-rajapintaa vaan on NoSQL-pohjainen (18). Muut mainitut kirjastot ovat ORM (Object-Relational Mapping) -kirjastoja. ORM on tekniikka, jolla olio-ohjelmoinnin oliot saadaan muutettua tietokannan ymmärtämään muotoon (19).

4 TIETOKANTAKIRJASTON UUDISTAMINEN

4.1 Nykyinen SugarORM-tietokantakirjasto

Nykyisessä Polar Flow Android -sovelluksessa on käytössä avoimen lähdekoodin SugarORM-tietokantakirjasto, jota käytetään monen erilaisen datan, kuten käyttäjän asetusten ja harjoitusdatan, tallentamiseen paikallisesti Android-laitteeseen, jotta niitä voidaan käyttää ilman internet-yhteyttä ja jotta käyttäjän tekemät muutokset voidaan tarvittaessa synkronoida rannelaitteelle ja verkkopalveluun sovelluksen synkronointilogiikan mukaisesti. Sovelluksessa on käytössä kirjaston vuoden 2016 versio 1.5, eikä kirjastosta olla julkaistu uudempaa versiota sen jälkeen (20).

4.2 SugarORM-tietokantakirjaston aiheuttamat ongelmat sovelluksessa

Nykyiseen kirjastoon ei ole saatavilla päivityksiä eikä sitä kehitetä. Tämä voi aiheuttaa ongelmia tulevaisuudessa julkaistavien Android-versioiden kanssa, jos Androidin SQLite-rajapintaan tehdään muutoksia. Nykyisen kirjaston CRUD (Create, Read, Update, Delete) -operaatiot ovat hitaampia kuin monella muulla uudemmalla kirjastolla. Myös tallennustilan kulutus on hieman suurempaa johtuen niin kirjastosta kuin myös siitä, että nykyisessä tietokannassa on tallennettuna dataa, jota ei tarvitsisi olla siellä. Nykyisen tietokannan mallin muuttaminen tietokantamigraatioiden avulla on turhan monimutkaista. Nykyinen tietokantakirjasto ei myöskään tue sovelluksessa laajalti käytössä olevaa RxJava2-kirjastoa, joka helpottaa reaktiivisten datavirtojen avulla asynkronisen koodin kirjoittamista sekä tekee säikeistyksestä helpompaa. Kirjastossa ei myöskään ole tukea Kotlinille ja sen ominaisuuksille kuten esimerkiksi vuorottaisrutiineille (coroutine). Vuorottaisrutiinit ovat ominaisuus Kotlinissa, ja ne sallivat asynkronisen koodin kirjoituksen ja hoitavat säikeistyksen kevyemmin verrattuna perinteisiin säikeisiin. Nykyinen kirjasto ei myöskään tue Android Studion Instant Run -ominaisuutta, joka nopeuttaa sovelluksen kääntämistä kehitysvaiheessa.

Nykyisen kirjaston käyttöönoton toteutus nykyisessä sovelluksessa ei vastaa hyviä olio-ohjelmoinnin suunnittelumalleja, mikä vaikeuttaa uusien ominaisuuksien kehittämistä ja vanhojen ominaisuuksien jatkokehitystä sekä tekee kehityksestä virheherkkää. Nykyisessä sovelluksessa tietokantaan liittyviä operaatioita ei ole abstraktoitu esimerkiksi DAO (Data Access Object) -luokkien taakse, vaan tietokantaoperaatiota tehdään ympäri sovellusta, kuten käyttöliittymäluokissa. Tämä aiheuttaa tiukan liitoksen (tight coupling) nykyisen tietokannan ja muun sovelluksen välille, tehden esimerkiksi tietokantakirjaston vaihtamisen työlääksi. Nykyisellä tietokannalla toteutettujen ominaisuuksien paikallinen yksikkötestaus on myös vaikeaa, ellei mahdotonta. Riippuvuus tietokantaan vaatii instrumentaatiotestejä, joiden ajaminen on hitaampaa ja niihin tarvitaan fyysinen tai emuloitu Android-laite.

4.3 Uusi tietokantakirjasto

Uudeksi tietokantakirjastoksi valikoitui Room Persistence Library. Se on Googlen kehittämä ja osa Android Architecture Components -kokoelmaa. Googlen kehittämänä on oletettavaa, että kirjaston tuki ei ole päättymässä pian ja se tulee sopimaan hyvin yhteen muiden uusien Android-kirjastojen ja suunnittelumallien, kuten MVVM-mallin kanssa. Room tarjoaa abstraktiokerroksen Androidin SQLiten päälle, helpottaen tietokannan hallitsemista. (21.)

5 ROOM-TIETOKANTAKIRJASTON KÄYTTÖÖNOTTO

5.1 Gradle-määrittely

Jos Room-tietokantakirjastoa halutaan käyttää sovelluksessa, se täytyy ensin liittää projektiin. Tämä onnistuu Android Studion Gradle-liitännäisellä. Gradle on koontityökalu, jolla voidaan määrittää esimerkiksi riippuvuuksia ulkopuolisiin kirjastoihin.

Aluksi määritellään käytettävä Room-versio projektin build.gradle-tiedostossa. Kuvassa 3 esitetään version määrittely.

```
// Room Persistence Library
roomVersion = '2.2.1'
```

KUVA 3. Room-version määrittely projektin build.gradle-tiedostossa.

Määritellään myös Kotlin- ja Kotlinx.coroutines-versiot. Kotlinista otetaan käyttöön versio 1.3.50, jotta se on yhteensopiva uusimman kotlinx.coroutines-kirjaston kanssa. Kuvassa 4 esitetään versioiden määrittely.

```
// Kotlin
kotlinVersion = '1.3.50'
kotlinxCoroutinesVersion = '1.3.2'
```

KUVA 4. Kotlin- ja Kotlinx.coroutines-version määrittely projektin build.gradle-tiedostossa.

Seuraavaksi siirrytään sovelluksen moduulikohtaiseen build.gradle-tiedostoon, jossa varsinaiset Roomin riippuvuudet määritellään, kuten kuvassa 5 esitetään.

```
// Room Persistence Library
implementation "androidx.room:room-runtime:${roomVersion}"
kapt "androidx.room:room-compiler:${roomVersion}"
implementation "androidx.room:room-ktx:${roomVersion}"
implementation "androidx.room:room-rxjava2:${roomVersion}"
testImplementation "androidx.room:room-testing:${roomVersion}"
```

KUVA 5. Room-kirjaston riippuvuudet sovelluksen moduulitason build.gradle-tiedostossa

Kotlinin riippuvuutta ei tarvitse määritellä uudelleen version päivityksen takia, koska se on jo määritelty ja uudempi versio tulee automaattisesti käyttöön. Kotlinx.coroutines-riippuvuus tulee kuitenkin määrittää (kuva 6).

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-jdk8:${kotlinxCoroutinesVersion}"
```

KUVA 6. Kotlinx.coroutines-määrittely sovelluksen moduulitason build.gradle-tiedostossa.

Lopuksi otetaan käyttöön kotlin-kapt-annotaatioprossessori, jotta Kotlinin annotaatiot saadan käyttöön. Kuvassa 7 esitetään annotaatioprosessorin käyttöönotto.

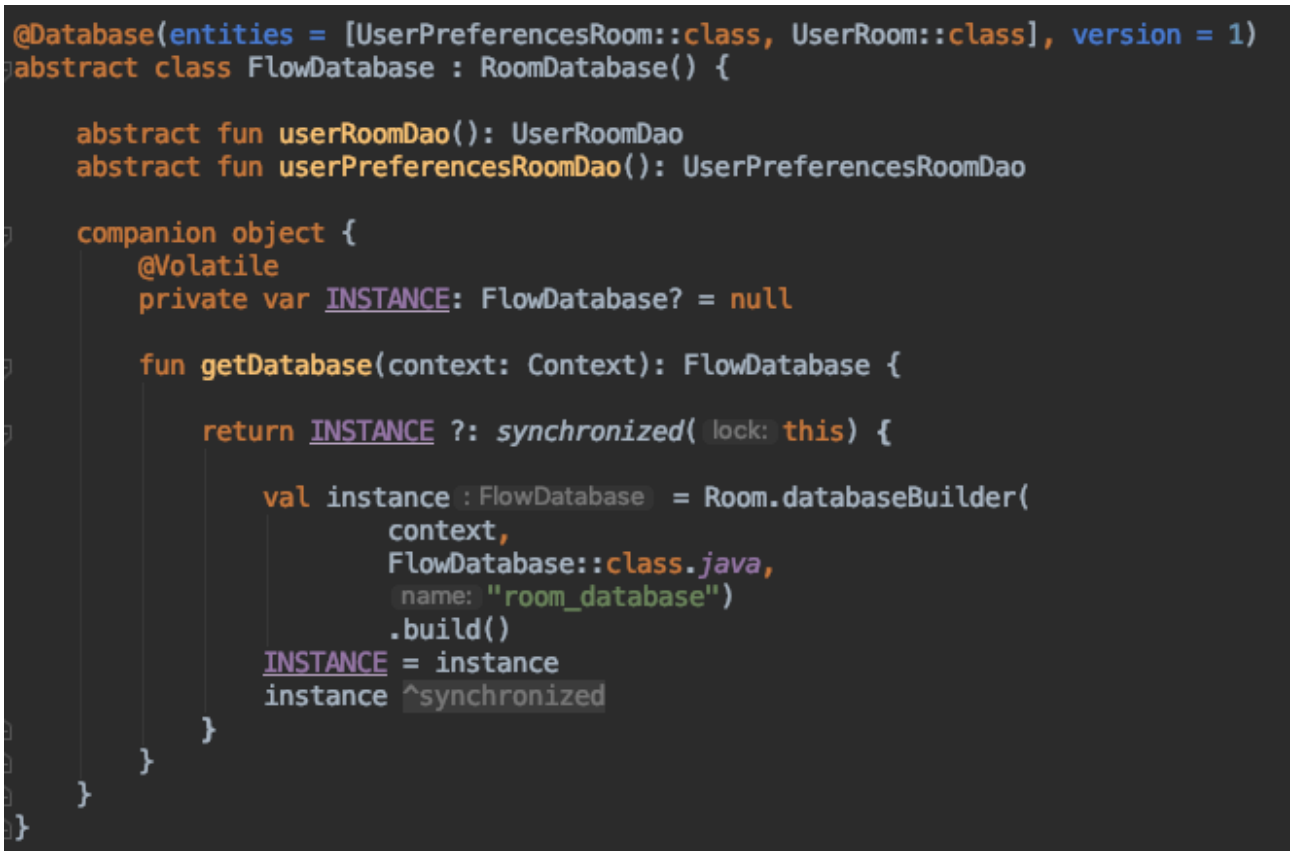
```
apply plugin: 'kotlin-kapt'
```

KUVA 7. Kotlin-kapt-annotaatioprosessorin käyttöönotto sovelluksen moduulitason build.gradle-tiedostossa.

Nyt kaikki tarvittavat Gradle-määrittelyt on tehty, ja seuraavan Gradle-synkronoinnin jälkeen määritellyt kirjastot ovat käytettävissä sovelluksessa.

5.2 Tietokannan määrittely

Seuraavaksi voidaan määritellä tietokantaluokka, kuten kuvassa 8 tehdään.



KUVA 8. Tietokantaluokan määrittely FlowDatabase.kt-tiedostossa.

Aluksi määritellään abstrakti luokka nimeltä FlowDatabase, joka perii RoomDatabase-luokan. Sille määritetään @Database-annotaatiolla tietokannan sisältämät entiteetit eli luokat, joista tehdään tietokantataulut tietokantaan, sekä tietokannan skeeman (schema) versio (22).

Seuraavaksi määritellään metodit userRoomDao() ja userPreferencesRoomDao(), joita kutsuessa palautuu siihen liitetty DAO-luokka. DAO-metodit ovat abstrakteja, sillä Room huolehtii niiden toteutuksesta (22).

Tämän jälkeen määritellään companion object. Companion objectin sisällä olevia jäseniä tai jäsenfunktioita voidaan kutsua staattisesti. Tämä tarkoittaa sitä, että luokasta ei tarvita ilmentymää (instance), vaan companion objectin jäseniä voidaan kutsua suoraan LuokanNimi.metodi()-funktiokutsulla (23).

Companion objectin sisällä määritellään INSTANCE-niminen kenttä, jonka tyyppi on FlowDatabase, ja sen alkuarvoksi annetaan null. Kenttä merkitään myös

@Volatile-annotaatiolla, mikä tekee kenttään tapahtuvat muutokset heti näkyviksi muissa säikeissä (24).

Companion objectin sisällä määritellään myös getDatabase() metodi, joka palauttaa INSTANCE kentän sisällön. Jos kentän arvo on null, luodaan uusi instanssi FlowDatabasesta RoomDatabase.Builder-luokkaa käyttäen. FlowDatabasesta voi olla kerrallaan vain yksi instanssi olemassa. Tämä toteuttaa olio-ohjelmoinnin Singleton-suunnittelumallin (25). RoomDatabase.Builder hyödyntää Builder-suunnittelumallia. Sitä voidaan käyttää helpottamaan olion luomisprosessia siinä tapauksessa, jos olio ottaa luomisvaiheessa vastaan paljon erilaisia parametreja ja niiden yhdistelmiä (26). Tässä tapauksessa builderille annetaan context, luokkaviittaus ja tietokannan nimi. Kun RoomDatabasen build()-metodia kutsutaan, luodaan tai avataan annetun niminen tietokanta, riippuen siitä, onko annetun nimistä tietokantaa jo olemassa. synchronized-funktio tekee tietokannan luomisesta thread-safen, tarkoittaen sitä, että tietokannan avaus tai luonti ei voi olla käynnissä kuin yhdestä säikeestä yhtä aikaa (27).

5.3 Taulujen määrittely

Kuvassa 9 määritellään UserPreferencesRoom-luokka, josta luodaan user_preferences -taulu.

```kotlin
package fi.polar.polarflow.userpreferences

import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.ForeignKey.CASCADE
import androidx.room.PrimaryKey
import fi.polar.polarflow.data.UserRoom

@Entity(tableName = "user_preferences", foreignKeys = [ForeignKey(
        entity = UserRoom::class, parentColumns = ["remoteIdentifier"],
        childColumns = ["remoteIdentifier"], onDelete = CASCADE)])
class UserPreferencesRoom {

    @PrimaryKey
    var remoteIdentifier: Long = 0

    var language: String? = ""
    var imperialUnits: Boolean? = false
    var twelveHourTimeFormat: Boolean? = false
    var firstDayOfWeek: Int? = 0
    var protoBytes: ByteArray? = byteArrayOf(0)

    constructor()

    constructor(language: String,
                imperialUnits: Boolean,
                isTwelveHourTimeFormat: Boolean,
                firstDayOfWeek: Int,
                protoBytes: ByteArray) {
        this.language = language
        this.imperialUnits = imperialUnits
        this.twelveHourTimeFormat = isTwelveHourTimeFormat
        this.firstDayOfWeek = firstDayOfWeek
        this.protoBytes = protoBytes
    }
}
```

KUVA 9. user_preferences-taulun määrittely UserPreferencesRoom.kt-tiedostossa

Tietokannan taulut luodaan luokista, jotka ovat annotoitu Roomin @Entity-annotaatiolla (28). Tässä tapauksessa @Entity-annotaatiolle annetaan parametrina myös luotavan taulun nimi ”user_preferences”, ilman tätä taulun nimeksi tulisi luokan nimi. Taululle määritellään myös viiteavain UserRoom-luokasta luotavan taulun remoteIdentifier-kentästä tämän taulun remoteIdentifier-kenttään. Tämä takaa viite-eheyden tämän taulun ja UserRoom-luokasta luotavan taulun välille (29). onDelete = CASCADE on vyörytyssääntö. Jos UserRoom-luokasta luotavasta taulusta poistetaan rivi, joka sisältää saman

remoteIdentifier-arvon kuin tässä taulussa oleva rivi, poistuu kyseinen rivi myös tästä taulusta (29).

Taulun kenttä remoteIdentifier asetetaan pääavaimeksi @PrimaryKey-annotaatiolla.

RemoteIdentifier on jokaisen Polar Flow -ekosysteemin käyttäjän yksilöllinen luku, jota käytetään käyttäjän tunnistamiseen.

Tauluun luodaan myös kentät: language, imperialUnits, twelveHourTimeFormat, firstDayOfWeek ja protoBytes. Huomattavaa on, että imperialUnits ja twelveHourTimeFormat määritellään Boolean-tyypiksi, mutta taulussa ne ovat Integer-tyyppiä, johtuen siitä, että SQLite ei tue Boolean-tietotyyppiä (30). Myös protoBytes-kenttä, joka on ByteArray-tyyppiä, tallennetaan BLOB-muodossa tauluun. BLOB (Binary Large OBject) sisältää puhdasta binääridataa (31). Kuvassa 10 on luotu user_preferences-taulu, joka sisältää kahden käyttäjän dataa.

	remoteIdentifier	language	imperialUnits	twelveHourTimeFormat	firstDayOfWeek	protoBytes
	Filter	Filter	Filter	Filter	Filter	Filter
1	42800922	fi	0	0	1	*BLOB*
2	43490220	en	1	1	3	*BLOB*

KUVA 10. user_preferences-taulu sisältäen kahden käyttäjän datan.

SQL-kielellä vastaava tyhjä taulu luotaisiin seuraavalla kyselyllä:

”CREATE TABLE `user_preferences` (`remoteIdentifier` INTEGER NOT NULL, `language` TEXT, `imperialUnits` INTEGER, `twelveHourTimeFormat` INTEGER, `firstDayOfWeek` INTEGER, `protoBytes` BLOB, PRIMARY KEY(`remoteIdentifier`), FOREIGN KEY(`remoteIdentifier`) REFERENCES `user`(`remoteIdentifier`) ON UPDATE NO ACTION ON DELETE CASCADE)”.

Myös user-taulu luodaan. Tauluun tarvitaan vain käyttäjän yksilöivä luku, joka asetetaan pääavaimeksi. Tämä tarvitaan yksilöimään sovelluksen käyttäjät, sillä samalla laitteella voi olla monta käyttäjää. Kuva 11 esittää user-taulun luomiseen käytettävää ohjelmakoodia.

```
package fi.polar.polarflow.data

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "user")
class UserRoom {
    @PrimaryKey
    var remoteIdentifier: Long = 0
}
```

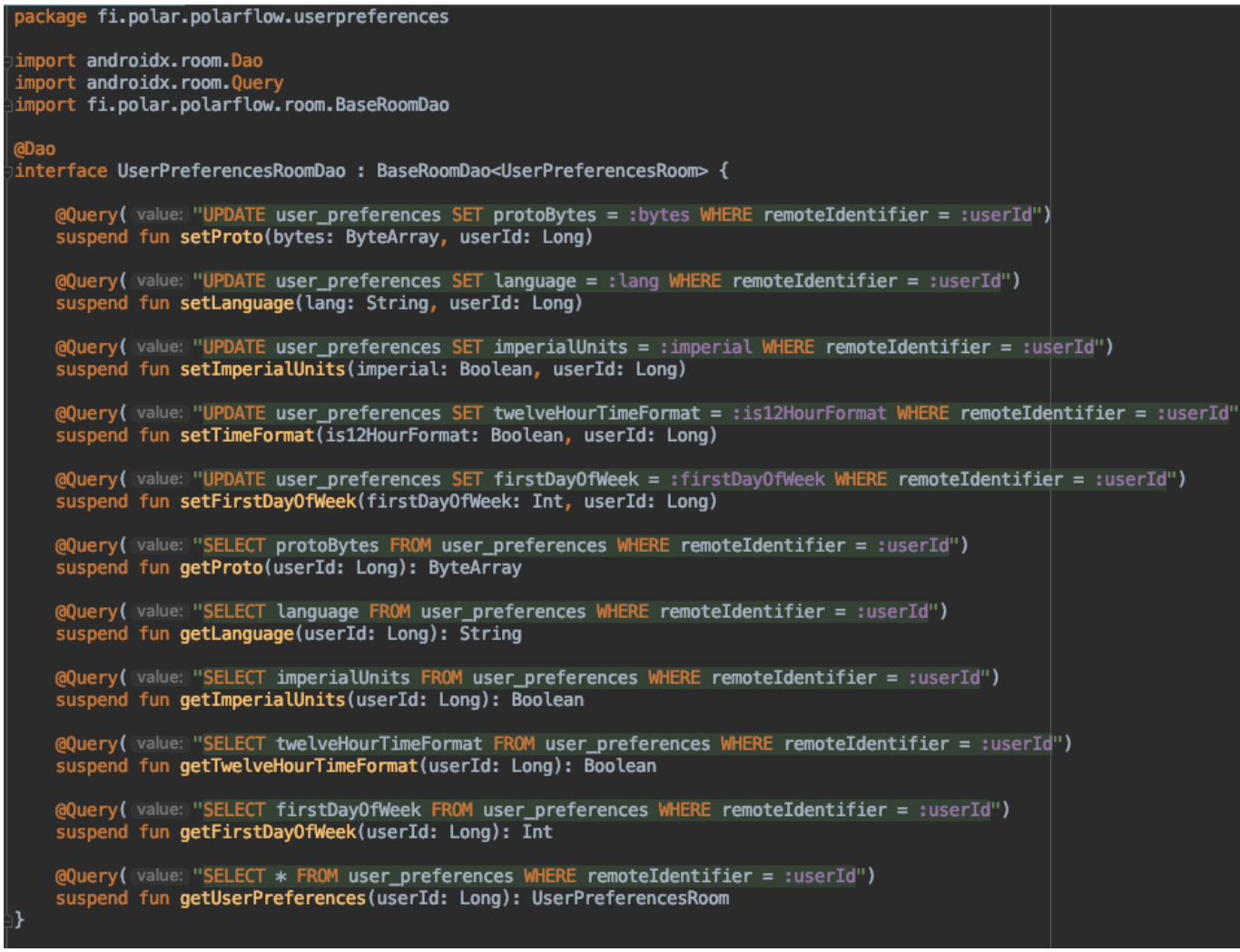
KUVA 11. user-taulun määrittely UserRoom.kt-tiedostossa.

Vastaava taulu luotaisiin SQL-kiellä seuraavalla kyselyllä:

”CREATE TABLE `user` (`remoteIdentifier` INTEGER NOT NULL, PRIMARY KEY(`remoteIdentifier`))”.

5.4 DAO-rajapintojen määrittely

DAO (Data Access Object) -rajapinnat huolehtivat tietokantaan tapahtuvista kyselyistä. Jokaiselle taululle tulee oma DAO-rajapinta selkeyden vuoksi. Room toteuttaa SQLite-luokat DAO-rajapinnoista (32). Kuvassa 32 on esitetty user_preferences-taulun käsittelyyn tarkoitettu UserPreferencesRoomDao-rajapinta.



```
package fi.polar.polarflow.userpreferences

import androidx.room.Dao
import androidx.room.Query
import fi.polar.polarflow.room.BaseRoomDao

@Dao
interface UserPreferencesRoomDao : BaseRoomDao<UserPreferencesRoom> {

    @Query( value: "UPDATE user_preferences SET protoBytes = :bytes WHERE remoteIdentifier = :userId")
    suspend fun setProto(bytes: ByteArray, userId: Long)

    @Query( value: "UPDATE user_preferences SET language = :lang WHERE remoteIdentifier = :userId")
    suspend fun setLanguage(lang: String, userId: Long)

    @Query( value: "UPDATE user_preferences SET imperialUnits = :imperial WHERE remoteIdentifier = :userId")
    suspend fun setImperialUnits(imperial: Boolean, userId: Long)

    @Query( value: "UPDATE user_preferences SET twelveHourTimeFormat = :is12HourFormat WHERE remoteIdentifier = :userId"
    suspend fun setTimeFormat(is12HourFormat: Boolean, userId: Long)

    @Query( value: "UPDATE user_preferences SET firstDayOfWeek = :firstDayOfWeek WHERE remoteIdentifier = :userId")
    suspend fun setFirstDayOfWeek(firstDayOfWeek: Int, userId: Long)

    @Query( value: "SELECT protoBytes FROM user_preferences WHERE remoteIdentifier = :userId")
    suspend fun getProto(userId: Long): ByteArray

    @Query( value: "SELECT language FROM user_preferences WHERE remoteIdentifier = :userId")
    suspend fun getLanguage(userId: Long): String

    @Query( value: "SELECT imperialUnits FROM user_preferences WHERE remoteIdentifier = :userId")
    suspend fun getImperialUnits(userId: Long): Boolean

    @Query( value: "SELECT twelveHourTimeFormat FROM user_preferences WHERE remoteIdentifier = :userId")
    suspend fun getTwelveHourTimeFormat(userId: Long): Boolean

    @Query( value: "SELECT firstDayOfWeek FROM user_preferences WHERE remoteIdentifier = :userId")
    suspend fun getFirstDayOfWeek(userId: Long): Int

    @Query( value: "SELECT * FROM user_preferences WHERE remoteIdentifier = :userId")
    suspend fun getUserPreferences(userId: Long): UserPreferencesRoom
}
```

KUVA 12. UserPreferencesRoomDao-rajapinta.

@Dao-annotaatiolla merkitään rajapinta DAO:ksi, jotta Room osaa generoida luokan, joka toteuttaa rajapinnan (32). @Query-annotaatiolla annetaan SQL-lause, jonka halutaan suoritettavan, kun määriteltyä metodia kutsutaan (33). Rajapinta perii myös BaseRoomDao<T>-rajapinnan. Rajapinta BaseRoomDao<T> hyödyntää Kotlinin Generics ominaisuutta, jossa voidaan antaa mikä tahansa tyyppi tyyppiparametrina (34). BaseRoomDao<T> sisältää yleisiä metodeja, joita voi käyttää uudelleen joka DAO-rajapinnassa, ilman että niitä tarvitsisi kirjoittaa uudestaan joka DAO-rajapintaan. Tämä rajapinta on esitettynä kuvassa 13. Suspend avainsana tekee metodista vuorottaisrutiinissa suoritettavan (35).

```
package fi.polar.polarflow.room

import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Update

interface BaseRoomDao<T> {
    @Insert
    suspend fun insert(obj: T)

    @Delete
    suspend fun delete(obj: T)

    @Update
    suspend fun update(obj: T)
}
```

KUVA 13. BaseRoomDao<T>-rajapinta.

5.5 Palveluiden määrittely

Vanhassa SugarORM-kirjaston käyttöönotossa tietokantaoperaatioita tehtiin ympäri sovelluksen koodikantaa. Tämä ei kuitenkaan ole suotavaa ja on selkeämpää käyttää yhtä luokkaa tietokantaoperaatioiden säilytyspaikkana. Sovelluksessa on jo osittain tehty näin verkko-operaatioiden kanssa. Vanhasta Volley kirjastosta on siirrytty uudempiin OkHttp/Retrofit-kirjastoihin ja tämän refaktoroinnin yhteydessä verkko-operaatioiden hallitseminen on siirretty palveluihin (Service). Nyt tietokantaoperaatiot siirretään myös palveluiden sisään. Esimerkiksi UserPreferencesService-palvelua käyttämällä voidaan asettaa tai hakea jokin user_preferences-taulun arvo, kuten kuvassa 14 esitetään.

```
isImperialUnits = BaseApplication.getModule()
        .getService(UserPreferencesService.class).getImperialUnits();
```

KUVA 14. imperialUnits-kentän arvon hakeminen UserPreferencesService-palvelua käyttämällä.

```
fun setImperialUnits(imperial: Boolean): Boolean {
    GlobalScope.launch { this: CoroutineScope
        withContext(Dispatchers.IO) { this: CoroutineScope
            launch { this: CoroutineScope
                dao.setImperialUnits(imperial)
            }
        }
    }
    return true
}

fun getImperialUnits(): Boolean {
    var result = false
    runBlocking(Dispatchers.IO) { this: CoroutineScope
        launch { this: CoroutineScope
            result = dao.getImperialUnits()
        }
    }
    return result
}
```

KUVA 15. UserPreferencesServicessä olevat asetus- ja hakumetodit imperialUnits-kentälle hyödyntäen vuorottaisrutiineja.

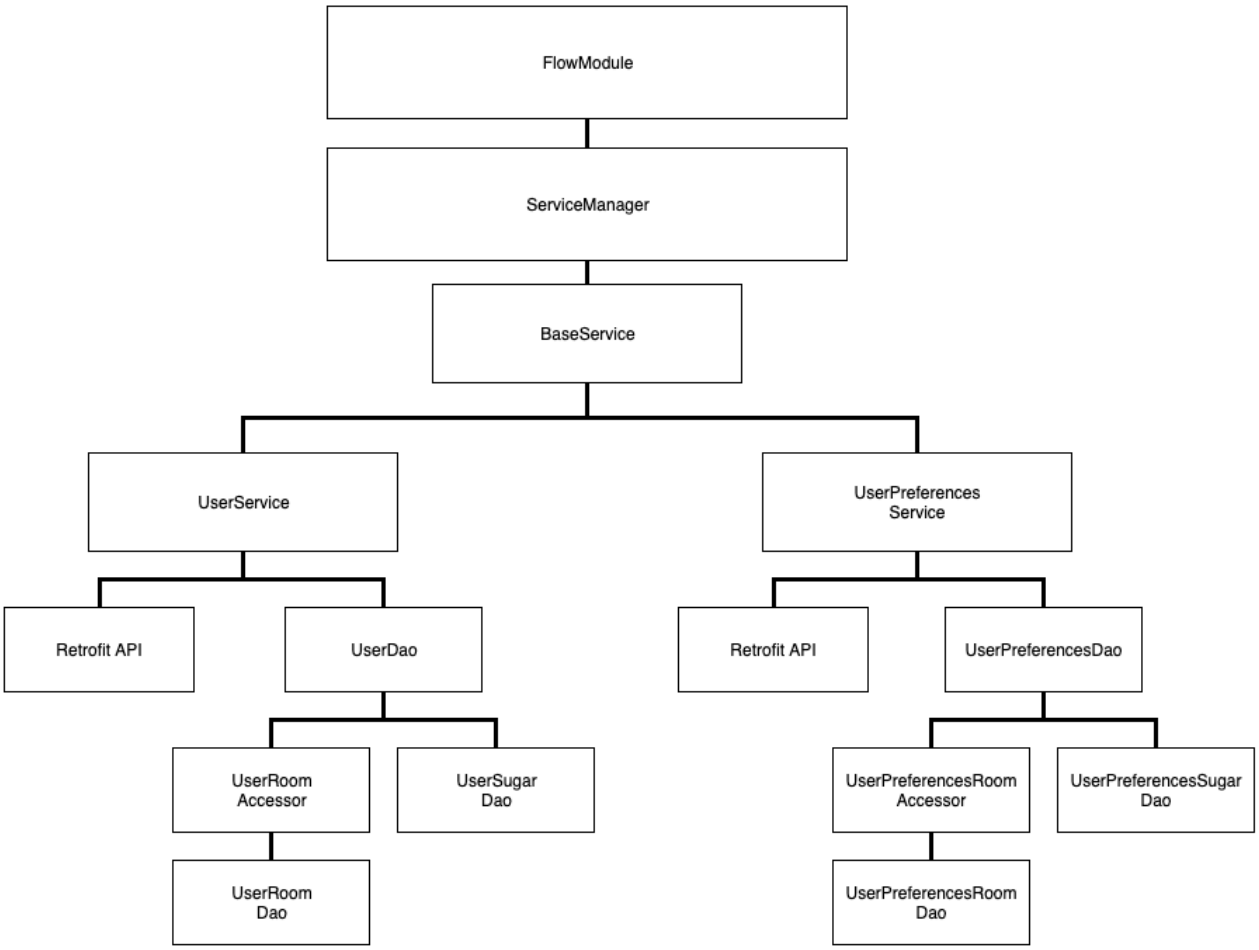
Tietokantaoperaatiot käynnistetään vuorottaisrutineissa CoroutineScopen launch-funktiolla, kuten kuvassa 15 esitetään. Tietokantaoperaatiot eivät sitten tule ajetuksi pääsäikeessä. Itse asiassa Roomissa on oletuksena päällä ominaisuus, joka ei salli tietokantaoperaatioden ajoa pääsäikeessä (36).

Palveluita hallitsee ServiceManager-luokka, jolla palvelut rekisteröidään sovelluksen käynnistyessä. Kuvassa 16 rekisteröidään UserPreferencesService-palvelu, jolle annetaan Retrofit-rajapinta ja UserPreferencesDao -rajapinnan toteuttava UserPreferencesRoomAccessor-luokka.

```
mServiceManager.register(UserPreferencesService.class,
        () -> new UserPreferencesService(mRetrofit.create(UserPreferencesApi.class),
                new UserPreferencesRoomAccessor(
                        FlowDatabase.Companion.getDatabase(context).userPreferencesRoomDao(),
                        BaseApplication.getModule().getService(UserService.class))
        ));
```

KUVA 16. UserPreferencesService-palvelun rekisteröinti ServiceManager-luokan oliolle FlowModule-luokassa.

Kuvassa 17 esitetään sovelluksen palvelupohjaista arkkitehtuuria.



KUVA 17. Sovelluksen palvelupohjainen arkkitehtuuri.

5.6 Tietokanta-agnostisuus

Tietokanta-agnostisuudella tarkoitetaan sitä, että sovellus tai palvelu ei tiedä, millä tietokantakirjastolla tai muulla toteutuksella tietoa haetaan (37).

```
class UserPreferencesService(private val api: UserPreferencesApi,
                             private val dao: UserPreferencesDao) : BaseService() {
```

KUVA 18. UserPreferencesService-luokan ja muodostinfunktion parametrien määrittely.

Kuten kuvassa 18 näkyy, UserPreferencesService-palvelu ottaa parametreiksi UserPreferencesApi ja UserPreferencesDao tyyppiset rajapinnat. UserPreferencesApi on Retrofit-rajapinta verkko-operaatioita varten. UserPreferencesDao on geneerinen rajapinta DAO-luokille, sen voi toteuttaa mikä tahansa eri tietokantatoteutus, tehden tulevaisuudessa tietokantakirjaston vaihtamisen helpoksi. UserPreferencesDao-rajapinta on esitettynä kuvassa 19.

```
package fi.polar.polarflow.userpreferences

interface UserPreferencesDao {

    suspend fun setUserPreferences(userPreferencesLocal: UserPreferencesLocal)

    suspend fun setProtoBytes(protoBytes: ByteArray)

    suspend fun setLanguage(language: String)

    suspend fun setImperialUnits(imperial: Boolean)

    suspend fun setTimeFormat(is12HourFormat: Boolean)

    suspend fun setFirstDayOfWeek(firstDayOfWeek: FirstDayOfWeek)

    suspend fun getUserPreferences(): UserPreferencesLocal

    suspend fun getProtoBytes(): ByteArray?

    suspend fun getLanguage(): String

    suspend fun getImperialUnits(): Boolean

    suspend fun getTwelveHourTimeFormat(): Boolean

    suspend fun getFirstDayOfWeek(): FirstDayOfWeek

    suspend fun initDefaultProto()

    suspend fun hasData(): Boolean
}
```

KUVA 19. UserPreferencesDao-rajapinta.

UserPreferencesDao:n toteuttaa tässä tapauksessa UserPreferencesRoomAccessor- sekä UserPreferencesSugarDao -luokat. UserPreferencesRoomAccessor sisältää Room-spesifiset toteutukset rajapinnalle. Se toimii myös sovittimena palvelun ja varsinaisen UserPreferencesRoomDao-rajapinnan välillä. Kuvassa 20 on esitettynä osa UserPreferencesRoomAccessor-luokasta.

```
class UserPreferencesRoomAccessor(private val roomDao: UserPreferencesRoomDao,
                                  private val userService: UserService) : UserPreferencesDao {

    override suspend fun setUserPreferences(userPreferencesLocal: UserPreferencesLocal) {
        setLanguage(userPreferencesLocal.language ?: "en")
        setImperialUnits(userPreferencesLocal.imperialUnits ?: false)
        setTimeFormat(userPreferencesLocal.isTwelveHourTimeFormat ?: false)
        setFirstDayOfWeek(FirstDayOfWeek.fromInt( value: userPreferencesLocal.firstDayOfWeek ?: 1))
        setProtoBytes(userPreferencesLocal.protoBytes ?: byteArrayOf(0))
    }

    override suspend fun getProtoBytes(): ByteArray {
        return roomDao.getProto(userService.getUserId())
    }

    override suspend fun setProtoBytes(protoBytes: ByteArray) {
        roomDao.setProto(protoBytes, userService.getUserId())
    }

    override suspend fun setLanguage(language: String) {
        roomDao.setLanguage(language, userService.getUserId())
        try {
            val locBuilder : Preferences.PbLocalizationPreferences.Builder = getBuilder()
            locBuilder.languageBuilder.setLanguage(language)
            val b : Preferences.PbGeneralPreferences.Builder! = Preferences.PbGeneralPreferences.newBuilder(
                    parseProto(getProtoBytes()))
            b.setLocalizationPreferences(locBuilder)
            b.lastModified = Utils.getPbNow()
            setProtoBytes(b.build().toByteArray())
        } catch (e: Exception) {
            FlowLog.e(TAG, string: "Failed to update language to proto: ", e)
        }
    }
```

KUVA 20. Osa UserPreferencesRoomAccessor-luokasta.

5.7 Protocol buffereiden käsittely

Polar Flow -ekosysteemissä on laajalti käytössä Googlen protocol buffer -serialisointiteknologia. Se on nopeampi kuin yleisesti käytetyt JSON- ja XML-serialisoinnit (38). Protocol buffereita käytetään sovelluksessa tiedon tallentamiseen ja synkronoimiseen rannelaitteelle ja verkkopalveluun. Protocol bufferit saadaan tässä tapauksessa tallennettua tietokantaan tavuina ByteArray-muodossa, josta Room muuttaa ne BLOB-muotoon. Tietokannasta haettaessa tavut saadaan parsittua haluttuun protocol buffer -muotoon. Tässä työssä refaktoroitiin sovellusta myös hieman löyhemmin kytketyksi käyttäjäasetuksia koskeviin protocol buffereihin.

5.8 Rinnakkaisuus vanhan kirjaston kanssa siirtymävaiheen ajan

Polar Flow Android -sovelluksessa on käytössä yli 100 eri tietokantataulua, joten refaktorointiprosessin voidaan olettaa kestävän pitkän aikaa. On mahdollista, että useampi tietokantakirjasto on käytössä yhtä aikaa, sillä tietokannat ovat erillisinä tiedostoina tiedostojärjestelmässä. Tietojen siirtäminen manuaalisesti tietokantakirjastosta toiseen ei ole järkevää, vaan on parempi antaa sovelluksen synkronointilogiikan huomata automaattisesti, että tauluissa ei ole tietoja, ja antaa sen hoitaa synkronointi tyhjään tauluun viimeiseksi päivitetystä tietolähteestä. Varsinaisesti SugarORM-kirjasto voidaan poistaa sovelluksesta vasta kun koko tietokannan toiminta on refaktoroitu käyttämään Roomia, ja voidaan olla varmoja, että kaikki synkronoimaton data on synkronoitu verkkopalveluun tai rannelaitteelle.

Tässä työssä tehtiin myös vanhan Sugar-koodin pohjalta erilliset DAO-luokat Sugarille, sillä koodin refaktorointi oli helpointa aloittaa niin.

5.9 Tietokantamigraatiot

Tietokantamigraatioilla tässä tapauksessa tarkoitetaan tietokannan skeeman (schema) eli taulujen sarakkeiden määrän, tyyppien tai nimien muuttamista olemassa olevaan tietokantaan. Tietokantamigraatioiden tekeminen on pyritty tekemään helpoksi Roomissa (39).

Esimerkiksi aiemmin esiteltyyn user_preferences-tauluun sarakkeen lisääminen onnistuu näin:

1. Lisätään UserPreferencesRoom Entity-luokkaan halutun tyyppinen ja niminen kenttä, tässä tapauksessa String-tyyppinen ja ”modified”-niminen kenttä, kuten kuvassa 21.

```
var modified = String
```

KUVA 21. Lisätty modified-kenttä UserPreferencesRoom-luokkaan.

2. Nostetaan tietokannan skeeman versio FlowDatabase-luokassa seuraavaan kokonaislukuun, kuten kuvassa 22.

```
@Database(entities = [UserPreferencesRoom::class, UserRoom::class], version = 2)
abstract class FlowDatabase : RoomDatabase() {
```

KUVA 22. Tietokannan skeeman version nosto versioon 2 FlowDatabaseluokassa.

3. Luodaan Room-migraatio-objekti, jolle annetaan versiot, mistä mihin migraatio tehdään ja suoritetaan SQL-kysely, jossa lisätään ”modified”-niminen TEXT-tyypin sarake ja alustetaan se merkkijonolla ”0”, kuten kuvassa 23.

```
private val MIGRATION_1_2: Migration = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL( sql: "ALTER TABLE user_preferences ADD COLUMN modified TEXT NOT NULL DEFAULT '0'")
    }
}
```

KUVA 23. Migraatio-objekti versioiden 1-2 migraatioon FlowDatabase-luokassa.

4. Lisätään migraatio-objekti tietokannan builderille, kuten kuvassa 24.

```
val instance : FlowDatabase = Room.databaseBuilder(
        context,
        FlowDatabase::class.java,
         name: "room_database")
        .addMigrations(MIGRATION_1_2)
        .build()
INSTANCE = instance
instance ^synchronized
```

KUVA 24. Seuraavalla kerralla tietokantaa avatessa tehdään versioiden 1-2 migraatio.

On myös mahdollista, että käyttäjä päivittää suoraan esimerkiksi versiosta 1 versioon 4. Tässä tapauksessa päivityksen jälkeen ajetaan migraatiot 1-2, 2-3 ja 3-4, kunhan ne kaikki ovat olemassa ja annettu builderille. Epäonnistuneet migraatiot aiheuttavat sovelluksen kaatumisen käynnistysvaiheessa. Tulevaisuudessa voi olla järkevää käyttää Roomin fallbackToDestructiveMigration()-metodia tietokannan rakennusvaiheessa, jolloin kaatumisen sijasta kaikki tiedot pyyhkiytyvät pois ja tietokannan taulut luodaan uudelleen. (39.) Tämän jälkeen tiedot synkronoituvat takaisin sovelluksen käyttöön verkkopalvelusta ja rannelaitteelta. Room tarjoaa myös

tuen migraatioiden yksikkötestaamiselle. Jokaiselle migraatiolle voi tehdä yksikkötestit, joissa luodaan tietokanta tietyn skeeman versiosta, asetetaan testidataa, suoritetaan migraatiot ja varmennetaan skeeman eheys.

Migraatioiden tekemisestä tehtiin myös dokumentaatiota Polarin sisäiseen Confluence-wikiin.

5.10 Tietokannan salaus

Salaamattomasta tietokannasta voi kuka tahansa paikallisesti laitteelle pääsevä ja riittävät käyttöoikeudet omistava taho lukea ja muokata tietoja. Käytännössä tämä tarkoittaa Android-laitteessa näytön lukituksen olemista avoinna ja root-käyttäjän oikeuksia. Androidissa käyttäjällä ei ole oletuksena root-käyttäjän oikeuksia, mutta ne ovat mahdollista saada niin sanotusti roottaamalla laite (40). Tietokannan salaus suojaa myös haittaohjelmilta ja muilta hyökkäyksiltä, joilla pyritään varastamaan käyttäjän dataa. Salaus ei estä tietokantatiedostoon pääsyä tai sen kopioimista, mutta siinä olevaa salakirjoitettua dataa ei saa muunnettua selkokielelle ilman, että salaus puretaan käyttämällä salausavainta, jota hyökkääjä ei tiedä.

5.11 Salaus Roomissa

SQLite-tietokanta voidaan salata SQLCipher-laajennusta käyttäen (41). Roomissa ei ole kuitenkaan tukea tällä laajennukselle, mutta tietokanta on mahdollista salata käyttämällä CWAC-SafeRoom-kirjastoa. CWAC-SafeRoom on osa CommonsWaren Android Components -kirjastokokoelmaa. SafeRoom toimii siltana Roomin ja SQLCipher-laajennuksen välillä. (42.) SafeRoom otetaan käyttöön Gradlen avulla, kuten kuvassa 25 esitetään.

```
implementation "com.commonsware.cwac:saferoom.x:${cwacSafeRoomVersion}"
```

KUVA 25. Gradle-määrittely CWAC-SafeRoom-kirjastolle. *Käytetty versio on 1.2.1.*

Lisätään myös CommonsWaren pakettivarasto Gradleen, jotta kirjasto osataan ladata sieltä, kuten kuvassa 26 esitetään.

```
repositories {
    mavenCentral()
    google()
    maven { url 'https://jitpack.io' }
    maven { url "https://s3.amazonaws.com/repo.commonsware.com" }
}
```

KUVA 26. CommonsWaren pakettivaraston määrittely Gradlella.

Tietokanta voidaan salata käyttämällä SafeRoomin SafeHelperFactory-luokkaa. Luokan muodostinfunktiolle annetaan salausavain joko tavu- tai merkkijonotaulukkona. Tämän luokan instanssi annetaan sitten tietokannan builderille openHelperFactory()-metodia käyttäen. Metodi korvaa oletuksena käytettävän FrameworkSQLiteOpenHelperFactoryn SafeRoomin SafeHelperFactorylla. (42.) Kuvassa 27 tehdään näin ja haetaan salausavain getEncryptionKey()-metodilla.

```
val instance : FlowDatabase = Room.databaseBuilder(
        context,
        FlowDatabase::class.java,
        name: "room_database")
        .openHelperFactory(SafeHelperFactory(getEncryptionKey()))
        .build()
```

KUVA 27. Tietokannan salaus SafeHelperFactorya käyttäen.

5.12 Salausavaimen suojaaminen

Helpoin tapa toteuttaa salaus olisi tallentaa käytettävä salausavain kovakoodattuna sovelluksen lähdekoodiin. Kuitenkin tämän salausavaimen löytäminen olisi mahdollista erilaisten takaisinmallinnustyökalujen (reverse-engineering) avulla. Yksi tällainen työkalu on Apktool, jolla salausavain voitaisiin purkaa helposti APK-tiedostosta (43).

Salausavaimen suojaamisessa voidaan kuitenkin hyödyntää Android Keystorea. Android Keystore on Androidissa oleva turvallisuusominaisuus, jossa voidaan säilyttää kryptografisia avaimia turvallisesti. (44.)

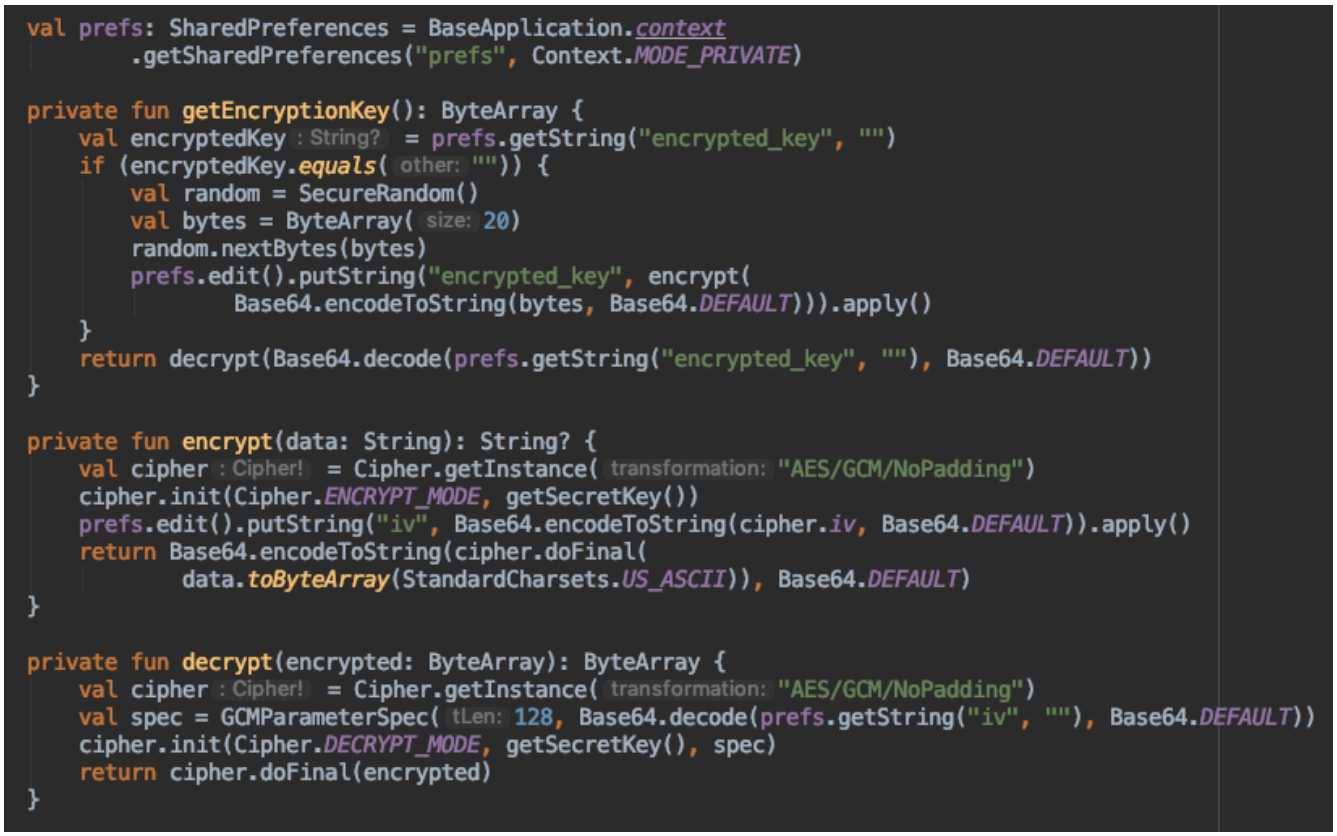
```
private fun getSecretKey(): SecretKey {
    val keyStore : KeyStore! = KeyStore.getInstance(
            type: "AndroidKeyStore").apply { load( param: null) }
    val secretKeyEntry : KeyStore.SecretKeyEntry? = keyStore.getEntry(
            alias: "room_database", protParam: null) as? KeyStore.SecretKeyEntry
    return secretKeyEntry?.secretKey ?: generateSecretKey()
}

private fun generateSecretKey(): SecretKey {
    val keyGenerator : KeyGenerator! = KeyGenerator.getInstance(
            KeyProperties.KEY_ALGORITHM_AES, provider: "AndroidKeyStore")
    val spec : KeyGenParameterSpec = KeyGenParameterSpec.Builder(
            keystoreAlias: "room_database", purposes: KeyProperties.PURPOSE_ENCRYPT
            or KeyProperties.PURPOSE_DECRYPT)
            .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
            .build()
    keyGenerator.init(spec)
    return keyGenerator.generateKey()
}
```

KUVA 28. getSecretKey()- ja generateSecretKey()-metodit FlowDatabase-luokassa.

Kuvassa 28 olevalla getSecretKey()-metodilla haetaan Android Keystoresta room_database nimellä oleva SecretKey. Jos room_database nimistä avainta ei ole, kuten sovelluksen ensimmäisellä käynnistyskerralla, luodaan avain generateSecretKey()-metodilla. Avainta luodessa määritellään käytettävä salausalgoritmi, avaimen tunnistenimi (alias), lohkosalaustila (block cipher mode) ja täytetila (padding mode). Saadusta SecretKey-oliosta voidaan kutsua Key-luokasta perittyä getEncoded()-metodia, jolla saadaan salausavain tavutaulukkomuodossa. Tämä ei kuitenkaan toimi Androidin Keystoren ollessa kyseessä, vaan getEncoded()-metodi palauttaa aina null arvon. Tämä johtuu Keystoren turvallisuusominaisuuksista, jonka vuoksi avainta ei saa sovellustasolla käyttöön (44).

Tämä voidaan kuitenkin kiertää antamalla SafeHelperFactorylle väliavain, joka salataan Keystoren SecretKeylla. Väliavain voidaan tallentaa Androidin SharedPreferences ominaisuutta hyödyntäen sovelluksen tallennustilalle. Tämän väliavaimen paljastuminen hyökkääjälle ei haittaa, koska se on salatussa muodossa ja sen purkamiseen tarvitaan Keystoren SecretKey.



```
val prefs: SharedPreferences = BaseApplication.context
        .getSharedPreferences("prefs", Context.MODE_PRIVATE)

private fun getEncryptionKey(): ByteArray {
    val encryptedKey : String?  = prefs.getString("encrypted_key", "")
    if (encryptedKey.equals( other: "")) {
        val random = SecureRandom()
        val bytes = ByteArray( size: 20)
        random.nextBytes(bytes)
        prefs.edit().putString("encrypted_key", encrypt(
                Base64.encodeToString(bytes, Base64.DEFAULT))).apply()
    }
    return decrypt(Base64.decode(prefs.getString("encrypted_key", ""), Base64.DEFAULT))
}

private fun encrypt(data: String): String? {
    val cipher : Cipher!  = Cipher.getInstance( transformation: "AES/GCM/NoPadding")
    cipher.init(Cipher.ENCRYPT_MODE, getSecretKey())
    prefs.edit().putString("iv", Base64.encodeToString(cipher.iv, Base64.DEFAULT)).apply()
    return Base64.encodeToString(cipher.doFinal(
            data.toByteArray(StandardCharsets.US_ASCII)), Base64.DEFAULT)
}

private fun decrypt(encrypted: ByteArray): ByteArray {
    val cipher : Cipher!  = Cipher.getInstance( transformation: "AES/GCM/NoPadding")
    val spec = GCMParameterSpec( tLen: 128, Base64.decode(prefs.getString("iv", ""), Base64.DEFAULT))
    cipher.init(Cipher.DECRYPT_MODE, getSecretKey(), spec)
    return cipher.doFinal(encrypted)
}
```

KUVA 29. SharedPreferences-kentän määrittely ja getEncryptionKey(), encrypt() ja decrypt() -metodit FlowDatabase-luokassa.

Kuvassa 29 olevalla getEncryptionKey()-metodilla tarkistetaan onko väliavainta jo luotu. Jos ei ole niin luodaan satunnainen tavutaulukko. Tavutaulukko muutetaan Base64-enkoodauksella binäärimuodosta merkkijonoksi, joka sitten salataan Keystorea käyttäen ja tallennetaan SharedPreferenceihin. encrypt()-metodissa tallennetaan myös Cipherin generoima IV (Initialization Vector) -arvo SharedPreferenceihin, koska sitä tarvitaan salauksen purkamisessa. IV tuo satunnaisuutta salaukseen, tässä tapauksessa yhtä IV-arvoa käytetään vain yhden kerran salaukseen, eikä arvon paljastuminen salauksen jälkeen heikennä salausta (45).

6 TESTAUS

Tämän tietokantauudistuksen testaus on tärkeää ennen sen julkaisemista valmiiseen tuotteeseen. Testauksella varmistutaan ohjelmiston virheettömyydestä. Muutoksia testataan manuaalitesteillä ja automaatiotesteillä. Manuaalitestausta tekevät kehittäjä kehitysvaiheessa, kehitystiimin testaajat sekä mahdollisesti yrityksen ulkopuolinen testauspalvelu. Manuaalitestauksessa muiden kuin ominaisuuden kehittäjän testaus on tärkeää, sillä kehittäjä ei ole välttämättä huomioinut kaikkia käyttötapauksia. Testaus siirtyy kehittäjältä muille testaajille, kun koodikatselmointi on suoritettu. Koodikatselmoinnissa muut kehittäjät perehtyvät muutoksessa tulleisiin koodimuutoksiin ja kommentoivat, jos koodista löytyy epäkohtia, jotka tulisi korjata. Koodikatselmointi suoritetaan höydyntäen Atlassian Bitbucket -ohjelmiston pull request ominaisuutta. Kun pull request on katselmoitu hyväksyttävästi ja testaus on suoritettu, voidaan tätä työtä varten tehdyn Git-versionhallintahaara liittää sovelluksen kehityshaaraan.

6.1 Testiautomaatio

Testiautomaatiolla tarkoitetaan sovelluksen testaamista automaattisesti ohjelmallisten testien avulla. Testiautomaatio auttaa kehittäjiä sovelluksen kehitysvaiheessa ja myöhemmin muutoksia tehtäessä. Yksikkötesteillä testataan sovelluksen yksittäisiä osia, kuten yhden funktion tai luokan toimintaa. Kehittäjä tekee uudelle ominaisuudelle yksikkötestit ja tarkistaa, toimivatko vanhat testit oikein koodimuutosten jälkeen. Polarilla on käytössä myös Jenkins-testiautomaatiojärjestelmä, joka kääntää sovelluksen APK-paketiksi ja suorittaa testit automaattisesti, kun versionhallintaan pusketaan uusia muutoksia.

6.2 Tietokantamuutoksien automaatiotestaaminen

Tässä työssä tehdyille tietokantamuutoksille voidaan tehdä monenlaisia testejä. Varsinaista tietokantaa ja sen RoomDAO-luokkia voidaan testata, palveluiden toimintaa voidaan testata sekä migraatioiden toimintaa voidaan myös testata. Tässä luvussa käsitellään tietokannan ja sen RoomDAO-luokkien testaamista. Tietokantaa testatessa tarvitaan instrumentoituja yksikkötestejä. Ne eroavat perinteisistä yksikkötesteistä niin, että niiden suorittamiseen tarvitaan fyysinen tai

emuloitu Android-laite, kun perinteiset yksikkötestit voidaan suorittaa suoraan paikallisessa Java-ajoympäristössä. Tässä tapauksessa tietokannan jäljittelemiseen tarvitaan Android-järjestelmää, joten testaus paikallisilla yksikkötesteillä ei onnistu.

```
class RoomUserPreferencesTest {

    private val userId = 1337L

    private lateinit var prefsDao: UserPreferencesRoomDao
    private lateinit var db: FlowDatabase

    @Before
    fun createDb() {
        val context : Context! = ApplicationProvider.getApplicationContext<Context>()
        db = Room.inMemoryDatabaseBuilder(
                context, FlowDatabase::class.java).build()
        prefsDao = db.userPreferencesRoomDao()
        runBlocking(Dispatchers.IO) { this: CoroutineScope
            launch { this: CoroutineScope
                db.userRoomDao().initializeUser(userId)
                db.userRoomDao().initializeUserPreferences(userId)
            }
        }
    }

    @After
    @Throws(IOException::class)
    fun closeDb() {
        db.close()
    }

    @Test
    @Throws(Exception::class)
    fun testSetAndGetLanguage() {

        // Arrange
        val language = "fi"

        // Act
        var result: String? = ""
        runBlocking(Dispatchers.IO) { this: CoroutineScope
            launch { this: CoroutineScope
                prefsDao.setLanguage(language, userId)
                result = prefsDao.getLanguage(userId)
            }
        }
        // Assert
        assertEquals(language, result)
    }

    @Test
    @Throws(Exception::class)
    fun testSetAndGetProtoBytes() {

        // Arrange
        val bytes : ByteArray = byteArrayOf(0x00, 0x01, 0x02, 0x03)

        // Act
        var result: ByteArray = byteArrayOf(0x7F)
        runBlocking(Dispatchers.IO) { this: CoroutineScope
            launch { this: CoroutineScope
                prefsDao.setProto(bytes, userId)
                result = prefsDao.getProto(userId)
            }
        }
        // Assert
        assertArrayEquals(bytes, result)
    }
}
```

KUVA 30. RoomUserPreferencesTest-testiluokka, jossa kaksi testiä.

Kuvassa 30 olevassa RoomUserPreferencesTest-luokassa @Before-annotaatiolla merkattu createDb()-metodi ajetaan aina ennen jokaista testiä.

Metodissa luodaan muistinsisäinen tietokanta ja alustetaan user- ja user_preferences- taulut. @After-annotaatiolla merkitty closeDb()-metodi ajetaan joka testin jälkeen. Metodissa suljetaan ja vapautetaan tietokanta muistista. @Test-annotaatiolla merkityt metodit ovat varsinaisia testejä. Niissä sijoitetaan alkuarvot, tehdään testattavia tietokantaoperaatioita ja lopuksi varmennetaan, että tietokannasta palautui odotettu arvo. Testiluokka ajetaan JUnit-testiautomaatio ohjelmistokehyksen versiota 4.12 käyttäen.

7 YHTEENVETO

Työn aiheena oli tehdä soveltuvuusselvitys SugarORM-tietokantakirjaston korvaamisesta Room-tietokantakirjastolla. Lopputuloksena saatiin aikaan esimerkkitoteutus Room-tietokantakirjaston käyttöönotosta Polar Flow App Android -sovelluksessa. Työstä on avattu versionhallinnan kautta pull request tietokantauudistuksen haarasta sovelluksen kehityshaaraan. Koodiin ehti tulla muutama katselmontikommentti, joiden perusteella on tehty pari pientä muutosta koodiin. Koodimuutokset ovat kuitenkin harvinaisen laajoja tässä pull requestissa ja sen katselmointi useamman kehittäjän tekemänä vie aikaa, joten koodikatselmointia ja mahdollisia muutoksia koodiin tuskin saadaan vietyä loppuun tämän työn aikana.

Tietokannan uudistamisen lisäksi sovellusta muutettiin löyhemmin kytketyksi käytettyyn tietokantakirjastoon. Käyttäjäasetuksiin liittyviä tietokantaoperaatioita ei tehdä enää käyttöliittymäluokissa, vaan käyttäjäasetuspalvelua käytetään tiedon muokkaamiseen. Palvelut tehtiin samalla myös riippumattomiksi käytetystä protocol buffer -serialisoinnista. Käyttäjälogiikkaa yritettiin myös refaktoroida Sugarista riippumattomaksi. Tosin tässä onnistuttiin vain osittain, koska osoittautui, että tämä on mahdotonta ennen kuin kaikki käyttäjään liittyvät tietokantataulut ovat refaktoroitu käyttämään Roomia. Tietokantaan toteutettiin myös salaus käyttäen CWAC-SafeRoom-kirjastoa ja avaimen hallintaan käytettiin Android Keystorea. Uudelle tietokannalle kirjoitettiin yksikkötestejä tietokannan testaamiseen. Polarin Confluence-wikiin tehtiin wikisivu Roomille, jonne dokumentoitiin, kuinka tietokantamigraatiot suoritetaan. Tehty työ tukee uutta MVVM (Model–View–ViewModel) -suunnittelumallia, jota on alettu käyttämään uusien ominaisuuksien teossa.

Kaiken kaikkiaan työ onnistui hyvin ja asetetut tavoitteet saavutettiin. Lopullista toteutusta ja tuotantoa tämä tietokantauudistus ei kuitenkaan ehdi nähdä tämän opinnäytetyön aikana johtuen ajallisista resursseista.

LÄHTEET

1. Platform architecture. Android Developers. Saatavissa: https://developer.android.com/guide/platform. Hakupäivä: 20.11.2019.

2. Sinhal, Ankit 2017. Closer Look At Android Runtime: DVM vs ART. AndroidPub. Saatavissa: https://android.jlelse.eu/closer-look-at-android-runtime-dvm-vs-art-1dc5240c3924. Hakupäivä: 20.11.2019.

3. HTC Dream 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/HTC_Dream. Hakupäivä: 21.11.2019.

4. Android Software Development 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Android_software_development. Hakupäivä: 18.12.2019.

5. Android application package 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Android_application_package. Hakupäivä: 21.11.2019.

6. Google Play 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Google_Play. Hakupäivä: 18.12.2019.

7. Java (programming language) 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Java_(programming_language). Hakupäivä: 22.11.2019.

8. Kotlin (programming language) 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Kotlin_(programming_language). Hakupäivä: 22.11.2019.

9. Lardinois, Frederic 2019. Kotlin is now Google's preferred language for Android app development. Saatavissa: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/. Hakupäivä: 25.11.2019.

10. Polar Flow app and compatible devices. Polar support. Saatavissa: https://support.polar.com/en/support/polar_flow_app_and_compatible_devices. Hakupäivä: 18.12.2019.

11. Polar Flow - Sync & Analyze 2019. Google Play. Saatavissa: https://play.google.com/store/apps/details?id=fi.polar.polarflow. Hakupäivä: 21.11.2019.

12. Database 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Database. Hakupäivä: 25.11.2019.

13. Laine Harri 2005. Tietokantojen perusteet. Helsingin yliopisto. Saatavissa: https://www.cs.helsinki.fi/u/laine/tkp/relaatiomalli/rakenne.html. Hakupäivä: 27.11.2019.

14. About SQlite. SQLite. Saatavissa: https://sqlite.org/about.html. Hakupäivä: 25.11.2019.

15. Most Widely Deployed and Used Database Engine. SQLite. Saatavissa: https://www.sqlite.org/mostdeployed.html. Hakupäivä: 25.11.2019.

16. SQL Features That SQLite Does Not Implement. SQLite. Saatavissa: https://www.sqlite.org/omitted.html. Hakupäivä: 26.11.2019.

17. Save data using SQLite. Android Developers. Saatavissa: https://developer.android.com/training/data-storage/sqlite. Hakupäivä: 25.11.2019.

18. Realm-java. Github. Saatavissa: https://github.com/realm/realm-java. Hakupäivä 25.11.2019.

19. Object-relational mapping 2019. Wikipedia. Saatavilla: https://en.wikipedia.org/wiki/Object-relational_mapping. Hakupäivä 25.11.2019.

20. Sugar releases 2016. GitHub. Saatavissa: https://github.com/chennaione/sugar/releases. Hakupäivä 21.11.2019.

21. Save data in a local database using Room. Android Developers. Saatavissa: https://developer.android.com/training/data-storage/room/index.html. Hakupäivä: 26.11.2019.

22. Database. Android Developers. Saatavissa: https://developer.android.com/reference/androidx/room/Database.html. Hakupäivä: 18.12.2019.

23. Objects and companion objects. Kotlinlang docs. Saatavissa: https://kotlinlang.org/docs/tutorials/kotlin-for-py/objects-and-companion-objects.html. Hakupäivä: 18.12.2019.

24. Volatile. Kotlinlang docs. Saatavissa: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-volatile/index.html. Hakupäivä: 18.12.2019.

25. Singleton pattern 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Singleton_pattern. Hakupäivä: 18.12.2019.

26. Builder pattern 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Builder_pattern. Hakupäivä: 18.12.2019.

27. Synchronized. Kotlinlang docs. Saatavissa: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/synchronized.html. Hakupäivä: 18.12.2019.

28. Entity. Android Developers. Saatavissa: https://developer.android.com/reference/androidx/room/Entity.html. Hakupäivä: 18.12.2019.

29. ForeignKey. Android Developers. https://developer.android.com/reference/androidx/room/ForeignKey.html. Hakupäivä: 18.12.2019.

30. Datatypes In SQLite Version 3. SQLite. Saatavissa: https://www.sqlite.org/datatype3.html. Hakupäivä: 18.12.2019.

31. Binary large object 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Binary_large_object. Hakupäivä: 18.12.2019.

32. Dao. Android Developers. Saatavissa: https://developer.android.com/reference/androidx/room/Dao.html. Hakupäivä: 18.12.2019.

33. Query. Android Developers. Saatavissa: https://developer.android.com/reference/androidx/room/Query. Hakupäivä: 18.12.2019.

34. Generics. Kotlinlang docs. Saatavissa: https://kotlinlang.org/docs/tutorials/kotlin-for-py/generics.html. Hakupäivä: 18.12.2019.

35. Composing Suspending Functions. Kotlinlang docs. Saatavissa: https://kotlinlang.org/docs/reference/coroutines/composing-suspending-functions.html. Hakupäivä: 18.12.2019.

36. Accessing data using Room DAOs. Android Developers. Saatavissa: https://developer.android.com/training/data-storage/room/accessing-data. Hakupäivä: 18.12.2019.

37. Database-agnostic applications 2015. dba-presents.com. Saatavissa: https://dba-presents.com/index.php/other/my-thoughts/34-database-agnostic-applications. Hakupäivä: 19.12.2019.

38. Gravell, Marc 2018. How are protocol-buffers faster than XML and JSON? Stackoverflow. Saatavissa: https://stackoverflow.com/questions/52146721/how-are-protocol-buffers-faster-than-xml-and-json. Hakupäivä: 19.12.2019.

39. Muntenescu, Florina 2017. Understanding migrations with Room. Medium.com. Saatavissa: https://medium.com/androiddevelopers/understanding-migrations-with-room-f01e04b07929. Hakupäivä: 19.12.2019.

40. Hildenbrand, Jerry 2019. Root Your Android Phone: What is Root & How To. Androidcentral. Saatavissa: https://www.androidcentral.com/root. Hakupäivä: 19.12.2019.

41. SQLCipher. zetetic.net. Saatavissa: https://www.zetetic.net/sqlcipher/. Hakupäivä: 4.12.2019.

42. CWAC-SafeRoom. Github.com. Saatavissa: https://github.com/commonsguy/cwac-saferoom. Hakupäivä: 4.12.2019.

43. Apktool. Ibotpeaches.github.io. Saatavissa: https://ibotpeaches.github.io/Apktool/. Hakupäivä: 4.12.2019.

44. Android keystore system. Android Developers. Saatavissa: https://developer.android.com/training/articles/keystore. Hakupäivä: 4.12.2019.

45. Pornin, Thomas 2011. Secret vs. Non-secret Initialization Vector. Stackoverflow. Saatavissa: https://stackoverflow.com/questions/5796954/secret-vs-non-secret-initialization-vector. Hakupäivä: 5.12.2019.